Towards a Theory for Integration of Mathematical Verification and Empirical Testing

Michael Lowry, Mark Boyd, Deepak Kulkarni
NASA Ames Research Center
M.S. 269-2, Code IC
Moffett Field, CA 94035
{lowry,mab,kulkarni}@ptolemy.arc.nasa.gov
http://ic.arc.nasa.gov/ic/projects/amphion/

Abstract

From the viewpoint of a project manager responsible for the V&V of a software system, mathematical verification techniques provide a potentially valuable addition to otherwise standard empirical testing. However, the value they add, both in terms of coverage and in fault detection, has been difficult to quantify. Potential cost savings from replacing testing with mathematical techniques cannot be realized until the tradeoffs can be quantified. This paper first describes a framework for a theory of software fault detection that is based on software reliability and formalized fault models. The novelty of this approach is that it takes into account the relative utility of the various tools for fault detection. Second, the ***paper*** *then describes a utility model for integrating mathematical and empirical techniques with respect to fault detection and coverage analysis for software. Third, the paper discusses how to determine the optimal combination of black-box testing, white-box (structural) testing, and formal methods in V&V of a software system. Finally, a demonstration of how this utility model can be used in practice is offered using a case study from a NASA software systems.*

1. Introduction

One of the Holy Grails of software engineering has been to find an effective, efficient, and cost-effective procedure for answering the question: How can one locate and fix 'the very last defect' in a piece of software? Since a perfect solution has yet to be found, the problem faced by software development project managers becomes one of optimization: How can one locate and fix as many as possible of the 'most important' defects in a piece of software given constraints on schedule, resources, and budget? The bottom line for software development managers is this: they want to know how to allocate their limited resources in order to find and remove as many defects as possible before the software is released, and in particular the defects that are most important to the requirements of their project.

Our long-term objective is to develop an expert advisory system for software engineers and project managers who are responsible for choosing tools and methods for software system Verification and Validation (V&V). V&V costs have always dominated development costs within the aerospace community, and are increasingly dominating development costs in all large-scale software projects in the general software development community. For example, Microsoft has allocated two in-house testers for every developer for its Windows NT 5.0 project - doubling the ratio from the Windows NT 4.0 project. Yet, no systematic method exists for choosing among the wide variety of V&V tools and methods offered by various commercial software-development tool vendors and academic researchers. Furthermore, little guidance has been offered on how to combine V&V methods to achieve a synergy that is greater than the sum of the value of applying individual methods.

The practical objective of developing a decision aid for V&V frames the case studies and conceptual development we report in this paper. Although we employ mathematics from various disciplines, such as the Bayesian statistical perspective found in the software reliability community, this work is independent of many of their controversial philosophical underpinnings. For example the quantitative measure of software reliability is controversial, because unlike hardware whose failure is considered to be determined primarily by wear-out of physical components over time, software failure is considered to be determined solely by latent design defects. Modeling failure rates due to latent design defects is controversial.

In our framework we do not postulate an absolute measure of software reliability. Rather, we focus on the differences in the derivatives of various software reliability growth models for different verification methods. Thus we use these models for their comparative predictions, as opposed to their absolute predictions.

With respect to formal methods, in our framework we do not postulate that they yield 'correct software', but rather that they provide effective methodologies for covering limited properties of *projections* of a software system. More than one method might need to be used in order to expose

0-8186-8750-9/98 $10.00

difficult defects, because the mechanism of a defect might span multiple projections.

Ultimately, then, our objective is to provide a decision aid for managing software V&V that can answer the following kinds of questions:

1. How long should software be tested through black-box simulation of expected operational use before switching to more sophisticated methods for V&V?

2. How can different verification methods be synergistically combined to detect particularly difficult types of defects?

3. How can testing be used to discharge assumptions underlying a formal methods verification, particularly when the assumptions can not be discharged through analytical methods?

4. How can formal analysis be used to transform a testing problem requiring excessive computational resources (trials) to a testing problem requiring far fewer resources?

1.1 Benefits of Integration

There are currently two major approaches to software verification and validation (V&V): traditional software testing techniques, and formal methods. Both of these approaches are limited by computational complexity in the size and/or scope of the analyses for which they may be used. Fortunately, the two methods break down due to computational complexity in ways which are very different and which are in some sense orthogonal. Specifically, black-box testing is limited in its ability to effectively cover known fault modes, due to its nature of random sampling. White-box (structural) testing, which often reduces to systematic sampling based on some coverage criteria (often related to a fault mode); has better coverage properties but still lacks the generality that comes from the symbolic calculation of formal methods. Formal methods are limited by the computational complexity of symbolic calculation, and often by the cost of the expert human labor required to use them. Formal methods and white-box testing are limited in their ability to detect unknown (i.e. unanticipated) fault modes.

As a consequence of these differences in coverage limitations between the types of V&V methods, it is possible to integrate the use of testing with formal methods in such a way as to allow the strengths of each to complement and offset the weaknesses of the other. The remainder of this paper discusses a framework for achieving this synergy between the two methods. Section 2 describes a framework for integration, and section 3 presents a NASA case study, in which the framework is developed into a mathematical utility model. Section 4 describes how to generalize the model. A longer technical report describes further case studies [4] and generalizations.

2. Integration of Empirical Testing and Formal Methods

We begin with a brief overview of both software testing and formal methods as software V&V methods, in order to describe the underlying nature of the mechanism each uses to uncover faults (latent errors) in the software and under what circumstances the mechanism of each breaks down.

2.1 Overview and Tradeoffs of Traditional Software Testing Methodologies

Traditional software testing methodologies have been empirical in nature, and generally fall into one of two major categories: *black-box* (functional-only) testing, or *white-box* (structural) testing. For black-box testing, the software is considered to be a process about which nothing is known of the internal structure or workings of the architecture or code. Consequently, only the functional performance, the inputs, the outputs, and the execution time and memory demands of the software are considered in an analysis. This type of testing tends to take the form of a protocol based strictly on statistical sampling of the input-output space of the software. In this paper our mathematical analysis assumes that this sampling is random, at least with respect to any particular fault mode for software failure.

In contrast to black-box testing, white-box testing permits details of the software's internal architecture and code to be taken into account. Tests are selected to cover some measure, such as the number of source code statements executed at least once. In this paper our mathematical analysis assumes that the tests are systematic with respect to one specified fault mode, but that the tests have limited effectiveness outside this fault mode.

For both black-box and white-box testing, a variety of strategies may be employed to dynamically estimate the effectiveness of the testing protocol in detecting faults, and thereby estimate the probability that additional undetected faults remain in the software after a certain amount of testing has been performed. An example of one such strategy is *fault seeding,* where a number of known faults are deliberately introduced into the code and the effectiveness of the testing is judged by observing how many of the seeded faults are detected by the testing protocol. Another, used in this paper, is through parameter fitting of software reliability growth models, based on the history of faults detected during testing.

A major strength of traditional black-box testing is that the tests are performed on the full software code in an execution environment which is identical or very close to the actual environment in which the software will be deployed. As a consequence, differences between the performance of the software during testing and the performance of the software

in the field are minimized. Furthermore, through techniques based on the *operational profile* [7], test resources can be focused on expected usage.

The major weakness of traditional testing methods is that they are fundamentally based on statistical sampling of the execution of the software. As a consequence, there is no way to guarantee that all faults will be uncovered, nor is there a way of guaranteeing that serious latent faults initially present in the software will have been uncovered after a specific amount of testing. Indeed some of the perennial problems that software testing managers face are such questions as: How long should testing go on until "enough" of the faults have been found and fixed? How can one be sure that a very serious fault did not evade the testing regime? Because of the hit-or-miss nature of empirical testing methods and the enormous size of the possible behavior space of the typical software system, the time required for testing increases dramatically as the desired fault coverage increases.

For high-confidence systems, the desired mean-time between failures often exceeds tens of millions of hours; thus empirical testing even according to the operational profile is insufficient to achieve this level of reliability. Therefore, the primary mechanism by which traditional empirical testing methodologies break down is the limitation inherent in statistical sampling of complex behaviors. This is illustrated by the Pentium floating point defect: under rare circumstances the floating point circuitry of the early Pentiums produced an incorrect result. These circumstances were not revealed during the extensive testing prior to the Pentium's release.

2.2 Overview and Trade-offs of Formal Methods

Unlike traditional testing, which samples the behavior of a digital system through observation of the system in action, formal verification mathematically calculates the behavior of a digital system. Note that the Pentium defect would have been revealed through formal verification algorithms. Subsequent to this costly defect, the digital hardware community has incorporated many formal verification algorithms into their design cycle.

To date, two main approaches to formal verification have been developed: computer-assisted theorem proving, and model checking. Computer-assisted theorem proving has the advantage of being capable of verifying unbounded (i.e., infinite state) systems. The theorem-proving approach has the disadvantage that it requires sustained effort over a substantial period of time on the part of a human expert. It also has the disadvantage that failure to derive a proof does not necessarily mean the software has a fault. While computer-based theorem provers have become increasingly powerful [8], the problem of finding suitable invariants for induction proofs will prevent them from being completely automated. Because of the difficulty of quantifying the cost of an expert's theorem-proving efforts, and the lack of the prospect for automation which could facilitate such a quantification of cost, we will not address this type of formal verification in our analysis. However, we do include limited forms of automated inference, e.g., partial evaluation.

The second major approach to formal verification is *model checking*, which is a mathematical technique for verifying and debugging concurrent or real-time systems modeled as interacting finite state machines. The disadvantage of model checking compared to theorem proving is that model checkers can only verify or debug systems of bounded size. The advantage of model checking over theorem-proving is that model checkers are completely automatic, and when they detect a software fault they provide an error trace demonstrating how it would lead to a software failure. However, when applied to software systems they require the software to be abstracted and translated to the language of the model checker. Further considerations of model checkers as applied to software is addressed in [4].

2.3 A Framework for Integration of Formal Methods with Testing

Our framework for integration is a method for deriving composite V&V protocols. Each analysis method, as applied to a V&V task, has a number of component costs and a number of utility factors. Examples are, respectively, execution time required for the analysis, and fault coverage - the portion of the total behavior space that is probed such that a fault lying in a probed region is guaranteed to be detected. Different methods can be compared for cost versus utility. V&V protocols that optimize the tradeoff between cost and utility can then be derived.

As an example, consider fault coverage versus the total cost of human labor and computer time for executing an analysis technique. Figure 1 shows a generic graph of the relationship between cumulative cost and fault coverage for black-box testing, white-box testing, and formal methods. The general pattern is that the rate of cost increase as fault coverage increases is greatest for black-box testing, somewhat less (but still relatively large) for white-box testing, and lowest for formal methods. However, white-box testing incurs an initial structural analysis cost, and formal methods can require a substantial modeling cost. Both these techniques incur a higher start-up cost than black-box testing.

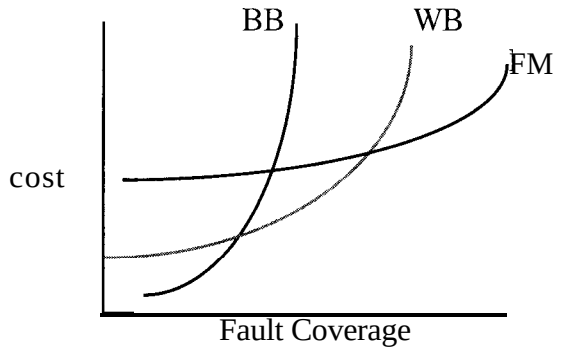


Figure 1: Relationship between Total Cost and Fault Coverage

In this paper, we will demonstrate our framework by deriving a composite protocol based on an incremental utility analysis: the expected computer time required to fmd the next software fault. Figure 2 shows a graph of this utility metric for these same methods, measured by the change in the failure intensity function (see section 2.3.1) versus total elapsed computer time. Formulas for the transition points where the incremental utility of one technique overtakes the incremental utility of another technique will be derived. The testing protocols will be parameterized on values which might not be known a priori, such as the size of a state space or the distribution of latent faults in a software system, and hence must be estimated during the protocol. An innovative feature of this protocol is that it explicitly considers different fault modes, and the relative strengths of different techniques for different fault modes. In the rest of this section we will overview the method and in the next section develop the mathematics, illustrated through a case study.

2.3.1 Deriving the Cost vs. Utility Curves for Empirical Testing

The curves for testing methods in Figure 2 are derived through software reliability growth models [6], which are models of the rate of reduction in failure intensity during the testing and debugging phases. A failure is a deviation from expected behavior of a software system; failure intensity is proportional to the number of software faults encountered per unit of execution time. (A software fault might or might not result in a failure when the code containing the fault is executed.) The models express the failure intensity as a function of either testing time or the cumulative number of faults detected. The models differ in their underlying assumptions about the fault intensity reduction process during testing and debugging, giving rise to different equations that express the failure intensity.

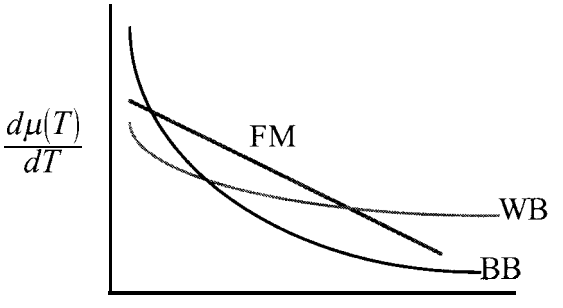


Figure 2: Rate for finding the next software fault versus cumulative execution time.

Two well-known software reliability growth models described in [6, pp. 30-50] are the Basic execution-time model and the logarithmic Poisson execution-time model. In the case study analysis presented in the following section, we will use the Basic model as part of the composite model for black-box testing. As compared to the Poisson model, the Basic model tends to favor testing over formal methods, and hence is more conservative. The failure intensity A, for the Basic model (expressed in terms of cumulative number of faults encountered rather than execution time) is given by:

$$\lambda_B(\mu) = \lambda_0 * \left(1 - \frac{\mu}{\nu_0}\right)$$

where λ_0 is the initial fault intensity at the start of testing, ν_0 is the total number of faults present in the software, and μ is the expected number of cumulative faults encountered at a given point in time during the testing regimen. These parameters are usually estimated using statistical fitting techniques on data collected during the testing and debugging of the software. The basic model assumes there are a finite number of faults, uniformly distributed, each of which is repaired effectively when it results in a failure during testing.

In our analysis, we are most concerned with the incremental cost of finding the next fault. In the basic model, the expected number of cumulative faults encountered versus execution time is a negative exponential approaching the limit of total latent faults:

$$\mu(T) = \nu_0\left(1 - e^{-\left(\lambda_0/\nu_0\right)T}\right)$$

The derivative of this function is the rate at which faults are encountered as a function of total execution time: $\frac{d\mu(T)}{dT} = \lambda_0 e^{-\left(\lambda_0/\nu_0\right)T}$

(The inverse of this derivative is a measure of the expected incremental execution time to find the next defect, and hence the incremental cost.) In this paper the Basic model is combined with a posterior Bayesian estimation model to derive the incremental utility function for black-box testing. The incremental utility function for white-box testing is based solely on a different posterior Bayesian estimation

model. The different Bayesian estimation models are due to considerations of the relative mechanisms of black-box versus white-box testing for different fault modes, as explained in section 2.4.

2.3.2 Deriving the Cost vs. Utility Curves for Formal Methods Analysis Techniques

The curve in Figure 2 for the formal methods technique, which in the case study for this paper is a symbolic algebraic analysis applied to partial evaluation of source code, is an estimate based on computational complexity and relative coverage. As in most formal methods techniques, it is assumed that the analysis is partitioned into a disjunction of symbolic cases. Each case is equivalent in coverage to a number of structure-based tests. The comparative utility of each case is the number of tests it subsumes divided by the expected computation time for the symbolic analysis. The cases can be rank-ordered according to this utility. The best cases are done first, leading to a monotonically decreasing function of the rate of fault discovery versus analysis time.

2.4 Theory of Defects: Fault Modes

The essence of our approach to integrating testing and formal methods is to provide a formal description of the fault modes by which latent software faults can cause software failures. For example, in the case study, the preconditions for an optimization routine are not satisfied. Specifically, the optimization routine requires a continuous function as one of its arguments - a function with any discontinuities in the interval of optimization will cause the optimization routine to fail. Known fault modes are associated with white-box (structural) test procedures and/or formal methods analysis methods that systematically uncover such faults. However, we recognize that not all fault modes will be known a priori. Therefore we include unknown fault modes in the calculations of reliability growth. Reliability growth is calculated separately for the different fault modes.

In the initial phase of black-box testing, discovered defects are attributed to one of these fault modes. The cross-over point for known fault modes between the utility of black-box testing and other methods is based in part on the difference between the effectiveness of random probes, with *replacement*, versus systematic procedures for uncovering faults. In the initial stages of testing, there is little difference between random probes with replacement versus probes without replacement. However, as testing proceeds, the probability of repeating a probe increases. Thus the utility of switching to systematic methods increases.

For unknown fault modes, black-box testing is presumed to be the only method for uncovering latent faults. By definition, the structural mechanisms by which unknown fault modes operate are not known a priori, and hence there is no systematic method based on structural considerations for detecting such faults. However, because only a fraction of discovered defects will be of a priori unknown fault modes, the failure intensity estimates for the unknown modes by themselves will be lower than failure intensity estimates for all fault modes combined. Furthermore, unless a large fraction of the defects are for unknown fault modes, the failure intensity estimates for unknown fault modes will be substantially lower compared to failure intensity estimates for all fault modes combined.

Thus, in the initial stages, black-box testing will be preferred because it uncovers both known and unknown fault modes. However, as testing proceeds, the failure intensity estimates for unknown fault modes will likely decrease rapidly, and the penalty for random probes with replacement for known fault modes versus systematic probes will increase. This determines the cross-over points, when the incremental utility of white-box or formal methods becomes greater than black-box testing. In principle, as V&V efforts continue, there may be cross-back points as the estimated failure intensity for the known fault modes is reduced vis-a-vis the estimated failure intensity for the unknown fault modes. We don't address these cross-back point calculations in our case study, but note that the basis for the calculations is the same as those for the cross-over points from black-box to structural/formal methods. In practice, the confidence intervals for the failure intensity maximum likelihood estimates will entail that the optimal V&V strategies be mixed, with the mixture ratio changing over time. In this paper we will concentrate on the trade-offs of the different verification methods as they relate to the different fault modes and leave the calculations of the confidence intervals to future research. We also leave to future research methods for dynamically increasing the number of known fault modes as debugging reveals the mechanisms of a priori unanticipated fault modes.

3. Case Study: Reusable Launch Vehicle

As part of the Reusable Launch Vehicle (RLV) condition-based maintenance project, a numerical optimization routine was used to estimate, based on sensor data, the amount of degradation in hardware components. The *Optimize()* routine that was used is part of the MatrixX package provided by ISI Inc., and is based on Karmarker's improved algorithm for convergence in constrained optimization. Convergence is guaranteed if the objective function, the constraint functions, and also their first derivatives are all continuous functions.

In this particular safety-critical RLV application, the objective function was a complicated program which included subroutines that computed factors requiring physical modeling ranging from a non-ideal gas law, to Newton's laws, to geometric constraints. A typical run of black-box testing of the application over the test suite typically took overnight,

and reviewing the logs took over an hour if the run was successful, and much longer if the run was not successful. After detecting an error, locating the source of the error (such as a continuity error) could take days, and each test-detect-locate-modify cycle took at least a week. After extensive testing and attempts at modification, lasting over several months, it was determined that the objective function could not be modified to reliably achieve the convergence conditions required for this particular safety-critical use of *Optimize()*.

An improved verification protocol would have greatly speeded up the testing and location of errors, thereby enabling the developers to explore more promising alternatives. In this section we contrast (admittedly after the fact) three methods for verifying and debugging this safety-critical software: black-box testing, white-box testing, and formal analysis. We will only consider these alternatives with the metric of fault detection, and not the metric of fault location. We note that additional considerations of fault location would favor white-box and formal analysis over black-box testing.

The *Optimize()* routine in MatrixX solves the following problem:

Find p such that $f(p)$ is minimized subject to:

$g(p) = 0$; $h_l <= h(p) <= h_u$; $p_l <= p <= p_u$

wherep is a vector of optimization parameters, with upper and lower boundsp, and p_u, $f(p)$ is the cost function, $g(p)$ is the equality constraint function, and $h(p)$ is the inequality constraint function with upper and lower bounds h_l and h_u,

A proof found in [3] proves the following theoretical result for the convergence of the *Optimize()* procedure. The proof requires that certain conditions on the cost function j hold for a correct application of the procedure. We have augmented these (proof) obligations with the abnormal predicate, thus providing the basis for both the known and unknown fault modes.

For all f , g , h , h_l, h_u, p_l, p_u continuous(f, p_l, p_u)
and continuous *(derivative@, p_l, p_u)*
and not(`abnormal`*(Optimize(f))*
=> *Optimize(f)* = p such that $f(p)$ is a global minimum over the interval p_l , p_u , and the region defined by the equality and inequality constraints.

3.1 Black-box testing

Because it tests the input-output behavior of a software system, black-box testing is a method for detecting unanticipated fault modes that are not part of the known fault modes. In the case study described in this section, these unanticipated modes are captured through the abnormal predicate. The known fault modes include those obtained through the negation of preconditions in the prelpost condition formalization of the routines in the system. Thus our comparison of black-box testing to other techniques involves two elements: first, a statistical probing of the known fault modes; and second, a statistical posterior estimation of the likelihood of unanticipated fault modes. If the errors found in the initial black-box testing phase fall within the known fault modes, then the posterior likelihood of unanticipated fault modes declines rapidly. In this paper we do not address the issue of the *prior* estimation of unanticipated fault modes, nor the augmentation of the known fault modes if new fault modes are discovered in black-box testing.

In this case study, a typical execution of the *Optimize()* routine may involve 10^4 or more executions of the cost function $f(x)$. In contrast, a white-box test for local continuity of the cost function and its derivative over a small neighborhood involves only a few executions of $f(x)$. Thus, white-box testing for the continuity conditions is orders of magnitude faster than testing the optimization routine itself, and the reliability growth that this fault mode (i.e. the failure of the *Optimize()* routine due to discontinuities) does not occur is much more rapid than through black-box testing.

We factor the analysis for reliability growth through black-box testing into two components. The first is a basic execution model for the reliability growth for unanticipated fault modes, corresponding to the abnormal predicate. (The basic execution model allows fault modes to be considered separately.) The second is an analysis of the effectiveness of black-box testing for the known fault modes of discontinuities in the cost function. For the known fault modes, we provide a more fine-grained Bayesian analysis than the basic execution model.

In the subsequent analysis we will compare the probability of finding the next fault through white-box testing of a small interval for discontinuity of the cost function f, to the probability of finding the next fault through an equal expenditure of cpu time in black-box testing. For the abnormal fault component uncovered through black-box testing, we will approximate the probability by multiplying the fault intensity (see section 2.3.1) by α, the number of execution units corresponding to the time it takes to execute one white-box test. This approximation is good as long as the product yields a probability much smaller than 1 .O:

$$\alpha * \lambda_0 * \left(1 - \frac{\mu}{\nu_0}\right)$$

The number of unanticipated faults is, by definition, not known a priori. This number is estimated after a period of debugging. The maximum likelihood estimate for the total number of abnormal faults based on the history of intervals between finding such faults, measured by cpu time during testing, is given by equation 12.41 in [6]:

$$\sum_{i=1}^{m_e} \frac{1}{\hat{v}_0 - i + 1} = \frac{m_e t_e}{\sum_{i=1}^{m_e} t_i + t_e(\hat{v}_0 - m_e)}$$

Where $\hat{v}_0$, the maximum likelihood estimate, is determined such that the terms on the left and right of the equality are equal; m_e is the number of abnorma l faults detected by time t_e; and in general the *ith* abnormal fault is detected at time t_i.

For the known fault modes, black-box testing is modeled as a random probe with possible repetition of successive probes. For this case study, assume that the *Optimize()* routine fails on any iteration of the inner loop that computes the cost function f at a point of discontinuity. Thus, every iteration of the inner loop includes a test for the continuity of the cost function f over an interval. However, this interval may be one that has already been probed. Let N denote the number of distinct intervals. If T intervals are already tested for continuity and if we assume the interval of the current iteration is randomly selected from all intervals, then the chance that the current interval is a (tested, untested) interval is, respectively: *T/N, (N-T)/N.*

The probability for testing T out of N intervals with M trials with replacement is given by the following recurrence relations for the function *s(M, T)* (where *s(M, T)* denotes the probability that exactly T distinct intervals are tested in M trials):

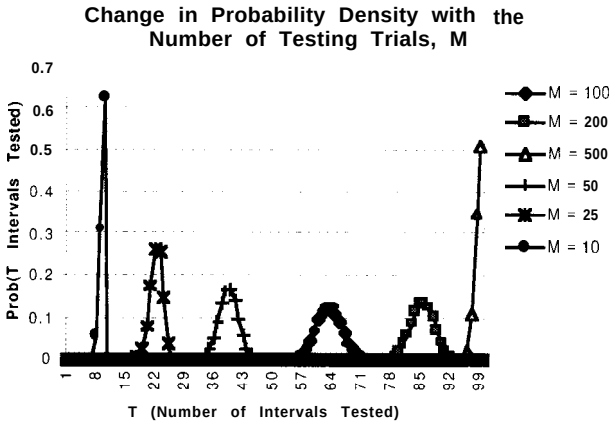
$$s(M,1) = N\left(\frac{1}{N}\right)^M$$

$$s(M+1,T+1) = \frac{(T+1)}{N} s(M,T+1) + \frac{(N-T)}{N} s(M,T)$$

In Figure 3 are several superimposed graphs for *s(M,T)* with N = 100, T varying between 1 and 100, and M varying from 10 to 500. As can be seen, the most likely estimate for T as a function of M is only slightly below M for small values, but obtaining full coverage of all intervals is very difficult. It takes 500 trials, just to get a 50% probability of full coverage.

In the next section on white-box testing, formulae will be derived for the *posterior* probability that the cost function is continuous as a function of the number of tested intervals *T*. The difference between white-box testing and black-box testing is that each call of f in white-box testing tests continuity over a new interval, whereas in black-box testing the chance it tests a new interval is *(N-T)/N*. Given the shape of this function, we can qualitatively see that in the initial stages of testing there is little difference between the reliability growth for the known fault modes between black-box and white-box testing. Furthermore, because black-box testing also probes unanticipated fault modes, the overall reliability growth will initially favor black-box testing. However, as testing proceeds, the gap between black-box testing and white-box testing for the known fault modes widens considerably, unless the unanticipated fault modes dominate.

Figure 3: Variation of Coverage against Cost (Measured by the Number of Trials)



3.2 White-box Testing

In contrast to the black-box tests, white-box tests for continuity will successively probe untested intervals ordered by some scheme. Hence every new test will check continuity of one new interval, and the number of tests M equals the number of tested intervals T.

To compare the incremental utility of black-box to white-box testing, we will need to consider posterior estimates for the number of both unanticipated faults (from eqn. 12.41 of [6]) and the known fault of discontinuity of the cost function or its derivative. The following analysis derives formulae for the posterior probability that the cost function f does not have any discontinuities.

Define the assertions *Obs, X* $_0$ and X_k as follows:

Obs, :T tests of distinct intervals have not revealed any discontinuities.

X_0 : f does not have any points of discontinuity.

X_k : f has k points of discontinuity.

Then the posterior probability $P(X_0|Obs_T)$ that there are no discontinuities given that T tests have been passed is derived as follows:

$$P(X_0 \mid Obs,) = \frac{P(X_0)}{P(Obs_T)} * P(Obs_T \mid X_0)$$

(from Bayes Theorem)

where $P(Obs_T|X_0) = 1$ always (by definition).

We compute $P(Obs_T|X_k)$ as follows: Let *N-k* intervals be continuous out of a total of N intervals. The probability that the first chosen interval will be continuous will be *(N-k) / N*. The second interval will be chosen from N-l intervals of which *N-k-l are* continuous. So the probability that the second interval will be continuous will be *(N-k-l) / (N-1)*.

Therefore:

$$P(Obs_T \mid X_k) = \frac{(N-k)*(N-k-1)*(N-k-2)*\ldots*(N-k-T+1)}{N*(N-1)*\ldots*(N-T+1)}$$

$$= \frac{(\text{N-k})!\ \ (N-T)!}{N!(N-T-k)!}$$

And: $P(Obs_T) = \sum_{k=0}^{N} P(X_k) * P(Obs_T \mid X_k)$

Hence: $P(X_0 \mid Obs_T) = \frac{P(x_0)}{\sum_{k=0}^{N} P(X_k) * P(Obs_T I X_k)}$

This gives the probability that no discontinuities exist in any intervals given that *T* distinct intervals have been tested and have not revealed any discontinuities. The graph in Figure 4, calculated with Excel, is a plot of this posterior estimate as a function of *T,* assuming an a priori probability distribution for X_k given by 1/2^(*k*+1), e.g., where the a priori probability of no intervals with discontinuities is 112.

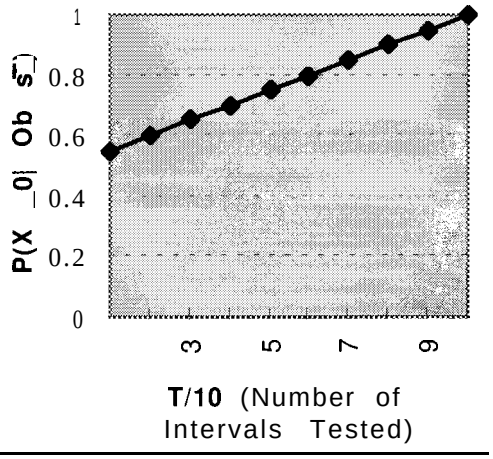


Figure 4: Posterior estimation of no fault discontinuities.

3.2.1 Optimal Cross-over for Black-Box and White-Box Testing

Given this posterior estimate, we now consider the optimal cross-over point between black-box testing and structural testing for the utility metric of the incremental cost for finding the next fault. To simplify the analysis, we only consider the scenario where there is either no intervals or exactly one interval with a discontinuity. We also assume that the number of intervals that are sampled in black-box testing is logged. This will enable a transition from black-box to structural testing without repeating the intervals already tested.

We will measure the incremental cost of finding the next defect as the probability that the next interval tested has a fault. For white-box (structural) testing, the only fault detected is a discontinuity in the cost function (or its derivative). For black-box testing, the faults include discontinuities and unanticipated faults.

For structural testing, the probability that the next interval finds a defect is the probability of there being a defect in some interval divided by the number of untested intervals:

$$\frac{1- P(X_0 \mid Obs_T)}{N - T}$$

For black-box testing, the probability is the union of the events of a discontinuity being detected and/or an unanticipated fault being detected. These are assumed to be independent events. We calculate this as the complement of the probability that neither type of fault is detected:

$$1-\left(\left(1-\frac{1-P(X_0 \mid Obs_T)}{N}\right)\times\left(1-\alpha\,\lambda_0 * \left(1-\frac{m_e}{\hat{\nu}_0}\right)\right)\right)$$

For small values of the probabilities of encountering a fault of either fault mode, we can approximate this through the following formula:

$$\left(\left(\frac{1-P(X_0 \mid Obs_T)}{N}\right)+\left(\alpha\,\lambda_0 * \left(1-\frac{m_e}{\hat{\nu}_0}\right)\right)\right)$$

Note that initially the number of tested intervals is 0, so the first term for black-box testing is equal to the probability of structural testing finding a discontinuity in the next interval (i.e., *N=N-T* initially). Thus black-box testing will be preferred initially, since the second term for unanticipated faults will provide an overall greater probability for finding a fault. The optimal crossover point from black-box to structural testing is approximated by the following equality, which is parameterized by the number of intervals tested *T,* and the estimates for finding unanticipated faults based on the cumulative history of such faults:

$$T\left(\frac{1-P(X_0 \mid Obs_T)}{N(N-T)}\right) = \left(\alpha\,\lambda_0 * \left(1-\frac{m_e}{\hat{\nu}_0}\right)\right)$$

3.3 Formal Methods

In many cases, the continuity of the cost function *f* can be proven algebraically over its entire domain, or at least partitioned into subintervals of the domain, some of which will be amenable to an algebraic proof of continuity. Here we outline the proof method and indicate how it can be combined with white-box testing for subintervals which are not amenable to algebraic proofs. As a simple example, consider the following program which computes *y* given x:

```
SUBROUTINE f (INPUT x: REAL): REAL
    LOCAL y: REAL;
    IF (x > 0) THEN y = x* x , ELSE y = x*x*x
    RETURN(y)
```

This subroutine partitions the real number domain into three intervals: $(-\infty, 0]$, $[0,0]$, $(0,\infty)$.

Over the first interval, the cost function f and its derivative are continuous if and only if $x*x$ and its derivative are continuous. Likewise over the third interval, the equivalent condition is on x*x*x. Over the second interval at the intersection point, we require the following two conditions: $x*x = x*x*x$ at $x=0$, and likewise for the first derivatives: $2x = 3x*x$ at $x=0$

This type of case analysis on intervals with algebraic proofs within intervals and at interval boundaries can be generalized through automatic program differentiation [1]. Roughly, the conditionals and case structure of the subroutine's source code will partition the input space into distinct intervals, most of which will be significantly larger than the small intervals tested through white-box testing. However, not all intervals will be amenable to algebraic proofs of continuity, and interval boundaries are often particularly difficult to analyze. Even if an interval can be analyzed symbolically, the cost of doing so might be computationally prohibitive.

To optimally combine structural testing with symbolic analysis, we assume that a symbolic analysis first partially partitions the input space for the cost function f into distinct intervals based on the top-level conditionals and case structure of the program. This partitioning might not be complete, since the general problem is intractable. Those intervals and interval boundaries which are identified are then sorted into a monotonically decreasing order according to the expected utility of symbolic analysis, defined as the equivalent number of white-box probes subsumed for an equivalent amount of cpu time. To be more concrete, consider the case where code differentiation has an expected cost in cpu time of N log N in the size of the algebraic term corresponding to a conditional branch within the program. Let W be the number of white-box probes that would be subsumed by the interval represented by the conditional branch. Then the optimal transition point between white-box testing and symbolic analysis in the ordered sequence is defined by:

$$N \log N \leq \alpha W$$

To summarize this case study, we have considered a NASA project where an approach to condition-based maintenance based on estimation through an optimization routine was ultimately found to not be reliable enough for the safety-critical requirements. A less costly approach to debugging than black-box testing would have led to this realization much sooner. Using our framework, we partitioned the fault modes into an expected class based on negating the preconditions of the optimization routine, and an abnormal class. We then analyzed three V&V methods: black-box testing, structural testing, and formal methods (algebraic analysis). Optimal crossover points based on the utility metric of the execution time to find the next fault were derived.

4. Conclusions

This paper described a framework for a theory of integrating different V&V methods based on software reliability growth models and formalized fault models. We have demonstrated through a case study that the utility of different approaches to V&V can be compared based on metrics such as the incremental cost of finding the next fault. The framework takes into account the relative strengths and weaknesses of black-box testing, structural testing, and formal methods. The latter are viewed as sophisticated debugging algorithms for certain classes of faults. A novel aspect of this framework is the abnormal fault mode, which enables differential analysis of anticipated and unanticipated fault modes. The advantage of black-box testing is its superior ability to uncover unanticipated faults, its disadvantage is its poor performance in detecting defects for anticipated fault modes. Structural testing and formal methods provide systematic and often efficient methods for uncovering defects for anticipated fault modes.

The framework postulates the existence of crossover points where an optimal V&V strategy would switch from one analysis method to another. Crossover points between black-box testing and structural testing; and between structural testing and symbolic methods, were derived. Further case studies and an extension of the analysis to model-checking are described in a longer technical report [4].

Although the mathematics in section 3 were developed in the context of a specific retrospective case study, the mathematics can likely be generalized as follows. Various mathematical models (including the Basic execution-time model) for software reliability growth based on testing have been extensively developed in the literature [5,6]. To apply our framework for calculating the tradeoffs for black-box testing, we require algorithms for hypothesis testing and maximum likelihood parameter estimations for these models. These algorithms, with associated software, can be found in [5], as well as case studies documenting how these models have been used in practice.

The specific structural (white-box) testing in this case study is based on partitioning the real-number line into subintervals. The size of each subinterval is small enough that a single test can determine the continuity of the derivative of a function over the subinterval. Nonetheless, the mathematical development in this case study generalizes to any method of partitioning the execution space of a software system. This includes not only various methods of partitioning the domain of the input parameters for the whole system or subsystems, but also partitioning methods based on paths through the code, decision points, function points, etc.

The key result is a method for calculating the tradeoff between black-box testing, which randomly samples the entire execution space; with more focused structural testing that

systematically samples each subset defmed by a partition. The key insight is that defining a fault mode, such as discontinuity of derivatives, enables a partitioning of the execution space into subsets. In this partition-based testing, it is assumed that testing an element of a subset will expose a defect over that subset within this fault mode.

The tradeoff with black-box testing arises because the same partition is unlikely to hold for other fault modes, particularly for fault modes that are unanticipated. Furthermore, it is assumed that focused tests designed to expose one particular fault mode will have limited effectiveness in exposing other fault modes. This leads to the tradeoff of black-box testing for testing multiple fault modes (known and unknown) versus systematic testing based on a partition for a particular fault mode. The mathematics for calculating this tradeoff reported here applies directly to any structural testing method for which there is a linear relationship between the amount of testing and the number of subsets of the partition which are sampled. The mathematics would need to be extended to handle structural testing methods that achieve a non-linear relationship between subsets sampled and the amount of testing, such as the MC/DC protocol [2] for test coverage of execution paths.

The analysis of formal methods reported here is extended in [4], which includes case studies that use model-checking and considerations of abstractions that reduce the computational complexity of applying these algorithms to software. It builds on previous work in [9] for combining pessimistic and optimistic testing strategies. The current state-of-the practice in formal methods as applied to software generally requires a substantial level of manual effort on the part of a formal methods expert. This is mainly due to the modeling effort required for automated methods such as model-checking, or to the exploration of different strategies in interactive methods such as verification theorem proving. However, various efforts are underway to make the application of formal methods far more automated in practice. Thus this paper has focused on a method for comparing the computational complexity of symbolic analysis versus structural testing. Like structural testing, symbolic analysis will typically partition the software analysis into a disjunction of distinct cases based on an anticipated fault mode. These different cases can be compared to the equivalent cost of analysis through white-box testing, and then each case can be analyzed through the most cost-effective method.

The contribution of this paper is in accordance with a developing perspective within the formal methods community, namely, that formal methods verification algorithms are primarily systematic debugging tools. We have taken this perspective to its next logical step, namely that of comparing the utility of testing to analytic algorithms for debugging. The comparison for any software system will depend on the particulars of the requirements of the software system (and hence the 'most important' defects to uncover), and the nature of the implementation. However, aspects of our framework will generalize, such as: the use of fault modes to compare black-box testing to systematic methods, the use of negated preconditions for subsystems to indicate anticipated fault modes, and the analysis of model abstraction for formal methods as it relates to false positives (see [4]). To achieve our ultimate objective of an expert advisory system for software project managers, future research will focus on the mapping from requirements/implementation to fault modes, and from fault modes to systematic methods for debugging.

5. Acknowledgments

We would like to thank the reviewers and our colleagues Jeremy Frank, Ann Patterson-Hine, Arthur Reyes, and Richard Sheldon for their insightful comments on preliminary versions of this paper.

6. References

[1] Bischof, C., and A. Griewank. "Tools for the automatic differentiation of computer programs", *in ICIAM/GAMM 95: Issue I: Numerical Analysis, Scientific Computing, Computer Science,* edited by G. Alefeld, 0. Mahrenholtz, and R. Mennicken, pp. 267-272.

[2] *DO-178B: Software Consideration in Airborne Systems and Equipment Certification.* Requirements and Technical Concepts for Aviation. Washington, D.C. December, 1992.

[3] Gill, P. E., Murray, W. and M. Wright. Practical Optimization, Academic Press, 198 1.

[4] Lowry, M., Boyd, M., and D. Kulkarni. "Towards a Theory for Integration of Mathematical Verification and Empirical Testing," NASA Ames Technical Memorandum, September 1998.

[5] Lyu, M. Handbook of Software Reliability Engineering, IEEE Computer Society Press, 1996.

[6] Musa, J., Iannino, A., and K. Okumoto. Software Reliability: Measurement. Prediction. Auolication, MacGraw-Hill, New York, 1987.

[7] Musa, J., Fuoco, G. Irving, N., and D. Kropfl. *"The Operational Profile,"* in Handbook of Software Reliability Engineering, M. Lyu (ed.), IEEE Computer Society Press, 1996.

[8] Rushby, J., and D. Stringer-Calvert. A *Tutorial for the PVS Specification and Verification Systems,* SRI-CSL-95-10, 1995.

[9] M. Young and R. Taylor. *"Rethinking the Taxonomy of Fault Detection Techniques,"* in Proceedings of International Conference on Software Engineering, 1989.